APPENDIX D

D.1 Example Assembler Shell Program for MS-DOS Interfacing

The Microsoft MACRO-86 assembler follows closely the Intel ASM-86 specifications. The operating system interfacing technique is via a straightforward interrupt (INT 21Hex), with the required operational parameter in the AH register. MS-DOS does not corrupt any registers other than the ones used for the sending or receiving of data. An example of the running and exiting program technique, plus the required assembler directives, follows. The program example is for the small memory model; but it will apply equally well to the compact or large memory model. The 8080 memory model is not recommended as it results in poor usage of the potential of the 8086/8088 processor. At link time, this programming example will generate an .EXE file - the header information on this file type will be found in E.1.

```
title   Example of MS-DOS/MACRO-86 Assembly Programming

dgroup  group   data
cgroup  group   code

msdos   equ     00021h                  ;interrupt to operating system

data    segment public   'data'
;######  insert your data here ######
data    ends

code    segment public   'code'
        assume  CS: cgroup, DS: dgroup

example proc    near                    ;origin of code

begin:
        push    ES                      ;save return segment address
        call    run_module              ;run the program
;
; run ends - select close down
;
exit    proc    far                     ;close down code
        xor     ax,ax                   ;zero for PSP:0
        push    ax                      ;save for far return
        ret                             ;and close down
exit    endp                            ;close down code ends

run_module:
        mov     ax,DATA                 ;get the data segment origin
        mov     DS,ax                   ; and initialize the segment
;##### insert your code at this point ######
        ret                             ;return to exit module

        example endp
        code    ends
        end
```

D.2 Example Assembler Shell Program for CP/M-86 Interfacing

The Digital Research ASM-86 assembler does not follow the standard Intel ASM-86 structure - this makes for a more complex task when transferring assembler programs between the CP/M-86 and the MS-DOS operating systems. The operating system interfacing technique is via a straightforward interrupt (INT EØHex), with the required operational parameter in the CL register. CP/M-86 corrupts all registers, excepting the CS and IP - it is, therefore, recommended that all registers be pushed prior to the INT EØHex being issued. An example of the running and exiting program technique, plus the required assembly directives, follows. The program example follows that of the MS-DOS MACRO-86 example. At GENCMD time, this programming example will generate a .CMD file - the header information on this file type is shown in the System Guide for CP/M-86.

```
title   'Example of CP/M-86/ASM-86 Programming'


reset   equ     ØØØØØh                  ;system reset function
cpm     equ     ØØØeØh                  ;interrupt to operating system

        cseg

begin:
        call    run_module              ;run the program
;
; run ends - select close down
;
        mov     cl,reset                ;select system reset
        mov     dl,ØØh                  ;select memory recovery
        int     cpm                     ;return to operating system
;
run_module:
;##### insert your code at this point ######
        ret                             ;return to exit module

        dseg
;##### insert your data here #####
        end
```

E.1 **MS-DOS -- EXE File Header Structure**

The Microsoft linker outputs .EXE files in a relocatable format, suitable for quick loading into memory and relocation. EXE files consist of the following parts:

- o Fixed length header
- o Relocation table
- o Memory image of resident program

A run file is loaded in the following manner:

- o Read into RAM at any paragraph (16 byte) boundary
- o Relocation is then applied to all words described by the relocation table.

The resulting relocated program is then executable. Typically, programs using the PL/M small memory model have little or no relocation; programs using larger memory models have relocation for long calls, jumps, static long pointers, etc.

The following is a detailed description of the format of an EXE file:

Microsoft .EXE File Main Header

Byte	Name	Function
Ø+1	wSignature	Must contain 4D5Ahex.
2+3	cbLastp	Number of bytes in the memory image modulo 512. If this is Ø then the last page is full, else it is the number of bytes in the last page. This is useful in reading overlays.
4+5	cpnRes	Number of 512 byte pages of memory needed to load the resident and the end of the EXE file header.
6+7	irleMax	Number of relocation entries in the table.
8+9	cparDirectory	Number of paragraphs in EXE file header.
A+B	cparMinAlloc	Minimum number of 16-byte paragraphs required above the end of the loaded program.
C+D	cparMaxAlloc	Maximum number of 16-byte paragraphs required above the end of the loaded program. ØFFFFh means that the program is located as low as possible into memory.
E+F	saStack	Initial value to be loaded into SS before starting program execution.
1Ø+11	raStackInit	Initial value to be loaded into SP before starting program execution.
12+13	wchksum	Negative of the sum of all the words in the run file.
14+15	raStart	Initial value to be loaded into IP before starting program execution.
16+17	saStart	Initial value to be loaded into CS before starting program execution.
18+19	rbrgrle	Relative byte offset from beginning of run file to the relocation table.
1A+1B	iov	Number of the overlay as generated by LINK-86. The resident part of a program will have iov = Ø.

The relocation table follows the fixed portion of the run file header and contains irleMax entries of type rleType, defined by:

rleType	bytes Ø+1	ra
	bytes 2+3	sa

Taken together, the ra and sa fields are an 8Ø86/8Ø88 long pointer to a word in the EXE file to which the relocation factor is to be added. The relocation factor is expressed as the physical address of the first byte of the resident divided by 16. Note that the sa portion of an rle must first be relocated by the relocation factor before it in turn points to the actual word requiring relocation. For overlays, the rle is a long pointer from the beginning of the resident into the overlay area.

The resident begins at the first 512 byte boundary following the end of the relocation table.

The layout of the EXE file is:

28-byte Header

Relocation Table

padding (<200hex bytes)

memory image

F.1 Victor 9ØØØ Technical Specification

Processor

- Intel 8Ø88 16-bit microprocessor
- 128k bytes RAM internally upgradeable to 896k bytes
- 4k bytes Auto-boot ROM (read only memory)
- 4 internal expansion slots for plug-in card options
- 2 x RS232C serial communications ports
- 1 x Parallel (Centronics) or IEEE-488 port
- 2 x Parallel user port (5Ø-way KK Connector on CPU board)

Display System

- 25 line x 8Ø column screen / 5Ø line x 132 column screen
- 12" CRT, Green p39 phosphor
- Adjustable horizontal viewing angle (± 45 degree swivel)
- Adjustable vertical viewing angle (Ø deg to 11 deg tilt)

Floppy Drives

- Standard 5 1/4-inch, single-sided 96 TPI dual disk drives, with a maximum capacity of 6ØØk bytes per drive.
- Optional 5 1/4-inch, double-sided 96 TPI dual disk drives, with a maximum capacity of 12ØØk bytes per drive.
- Optional single 1Ø,ØØØk byte Hard Disk - non-removable; with single 5 1/4-inch, double sided 96 TPI disk drive with a maximum capacity of 12ØØk bytes.

Single-sided floppy drive offers 8Ø tracks at 96 TPI
Double-sided floppy drive offers 16Ø tracks at 96 TPI
Floppy drives have 512 byte sectors; utilising a GCR, 1Ø-bit recording technique.

Floppy access times:
2 micro-second per bit data transfer rate, with an interleave factor of 3. Average seek time is approximately 9Ø milli-seconds.

Hard Disk access times:
Ø.2 micro-second per bit data transfer rate, with an interleave factor of 5. Average seek time is approximately 1ØØ milli-seconds.

Keyboard

Separate Intel 8Ø48 microprocessor
Fully software definable with 1Ø soft function keys
Full IBM Selectric III (56 key) keyboard layout
Type ahead buffering to 32 levels and full n-key rollover
Keyswitches rated for 1ØØ million operations

Electrical

Input voltage 9Ø-137 VAC or 19Ø-27Ø VAC (internal jumper)
Input frequency 47-63 Hz

Environment

Operating temperature Ø deg C to 4Ø deg C
Operating humidity 2Ø% to 8Ø% (non-condensing)
Storage temperature -2Ø deg C to 7Ø deg C
Storage humidity 5% to 95% (non-condensing)

F.2 Victor 9000 Physical Specifications

Mainframe Assembly

Height	Width	Depth	Weight (approx)
178 mm	422 mm	356 mm	12.6 kg
7 in	16.6 in	14 in	281 lbs

Display Assembly

Height	Width	Depth	Weight (approx)
264 mm	326 mm	339 mm	8.1 kg
10.4 in	12.9 in	13.4 in	18 lbs

Keyboard Assembly

Height	Width	Depth	Weight (approx)
45 mm	483 mm	203 mm	1.5 kg
1.8 in	19 in	6.4 in	3 lbs

System Assembly

Height	Width	Depth	Weight (approx)
457 mm	483 mm	559 mm	22.2 kg
18 in	19 in	20.4 in	49 lbs

Width without the keyboard module is 396 mm / 15.6 in

G.1 Glossary of Terms

The following table is a glossary of terms found in this manual:

BAUD	The term baud rate means the number of bits sent down a line per second. A baud rate of 300 will, therefore, be capable of transmitting data at 300 bits per second. Since a textual character is composed of 8 bits, then 37.5 characters could be sent per second at this baud rate.
BIOS	This means the Basic Input Output System. The BIOS is a fundamental portion of an Operating System, allowing the operating system to communicate correctly with any peripheral devices; typical BIOS modules include the disk driver; the keyboard input driver; the screen driver; the printer driver.
BIT	A bit is a binary digit. The bit can, therefore, contain either One or Zero. A One is bit HIGH or ON. A zero is bit LOW or OFF. A bit may be likened to a light-switch - the switch can only be on or off. See BYTE.
BOOT	This term comes from the phrase "the computer pulls itself up by its boot-strap". The term boot-strap means the same, but is no longer in such common use. To boot a computer is to load an operating system - the computer does this by means of a boot-strap program. The computer, when switched on, is not aware of its environment - but it automatically runs its boot-strap program. The Victor 9000 boot-strap program is stored in the boot PROM; it first causes the display of the little disk picture - it then searches for a disk with an operating system - when it finds this disk, it loads the operating system and begins to execute it. The boot-strap program is not used again until the reset switch is pressed, or the power is switched off and on.
BUS	A bus in computer jargon is not unlike a bus to carry passengers. When data is moved around inside a computer it is moved along the bus wires. These bus wires connect the Victor 9000 microprocessor to its memory, disk(s) and screen.

BYTE	A byte is a collection of 8-bits or two nibbles. A byte may store one character of text, or a number from 0 to 255 in binary.
DOT MATRIX	A printed character on the screen or a dot-matrix printer may be viewed as a square containing dots. On the Victor 9000 screen a character has a square cell (matrix) of 16 dots high by 10 dots wide - within this box, the dot on/off patterns create a viewable character.
FONT CELL	In reference to DOT MATRIX, the font cell is the collection of bytes of data that make up the character dots that are to be displayed on the screen. Each character on the screen is composed of pre-defined patterns of dots to make the viewed dot matrix. These patterns of dots are stored in the Victor 9000 memory as data - the screen controller chip scans these data bytes and the resulting character image is displayed on the screen.
HEADER	A header on a file gives information to the operating system on where and how the file is to be loaded in to memory. Many files provided by Victor Technologies (such as keyboard and character set files) contain headers that are not used by the operating system, but are used by Victor Technologies utilities.
INTERRUPT	An interrupt is some event occuring in the computers environment that the computer will stop all other activities for. An example of an interrupt is a key-press. If you press a key on the Victor 9000, an interrupt is generated; at this point the processor stores all information on its current task and gets and saves the value of the key pressed; it then picks up all the information it stored on its last task and continues where it left off. This whole series of events takes only a few micro-seconds.
NIBBLE	Sometimes spelled NYBBLE; a nibble is half a byte or 4-bits. See BYTE and BIT.
OPERATING SYSTEM	An operating system allows the computer to be aware of its environment and gives the user the ability to enter and retrieve data from the computer.

PROM	Programmable Read Only Memory, PROM, is a chip or collection of chips that is used to store permanently a single computer program or collection of computer programs. The boot-prom, sometimes called boot-rom, contains all the information the Victor 9000 computer needs to read an operating system from disk. There are different types of prom; EPROM which is erasable prom, simply shine a high-powered ultra-violet lamp on the chip, and it can be re-programmed; etc.
RAM	Random Access Memory, RAM, is a chip or collection of chips that is used to store temporarily (until the power is removed) data, computer program(s), text, etc. This is the memory of a computer.
REGISTER	A computer register is a portion of the processor. The Victor 9000 uses the Intel 8088 micro-processor - there are several different types of registers within this chip; there are 8-bit registers, and 16-bit registers. Data is generally not manipulated in RAM, but is brought in to a register of the processor and manipulated there, then the result saved from the register back into RAM.
WORD	A word is a number of bits, generally greater than 8. The Victor 9000 has a 16-bit word - thus a word in the Victor 9000 is composed of two bytes. The DEC PDP-8 computer has a 12-bit word - on this machine, therefore, a word is composed of one byte and one nibble.

H.1 MS-DOS Base Page Structure

The MS-DOS Base Page (sometimes called the Program Segment Prefix or PSP), is created when you enter an external command. COMMAND.COM will allocate a memory region to the external program, and will insert the Base Page prior to the origin of this program.

In the memory segment that the program is to load, COMMAND.COM places a Base Page, COMMAND.COM then loads the program at an offset of 100hex, and hands over control to the external program. The external program, once its function is complete, hands control back to the operating system by a far JUMP or far RETURN to location zero within the Base Page; the instruction at this location is an INT 20, or return control to MS-DOS. This stage must be executed to allow MS-DOS to recover memory correctly (see Appendix D.1).

When an external program is loaded, the following conditions are true:

> The file control blocks at Base Page locations 5Chex and 6Chex are created from the first two parameters entered on the command line.
>
> The command line at Base Page location 80hex is created from the command line entered AFTER the program filename. The byte at location 80hex contains the command line character count, the following bytes contain the raw command line as entered at the keyboard.
>
> The word at offset 6 in the Base Page contains the number of bytes available in the segment.
>
> The contents of register AX are established to reflect the validity of the drive(s) on the command line. Thus the following may be found:
>
> > AL = FFhex when the first drive letter on the command line was not recognized by MS-DOS.
> > AH = FFhex when the second drive letter on the command line was not recognized by MS-DOS.

The above applies equally to both .EXE and .COM type files. The .EXE and .COM files do have differences when the they load, and these are described more fully below.

When .EXE files load:

The contents of register DS and register ES are pointing at the Base Page segment address.

The registers CS, IP, SS and SP are initialized to those values passed by the linker.

When .COM files load:

The contents of registers CS, DS, ES, and SS are pointing to the Base Page segment address.

The register IP is set at 1ØØhex.

The register SP is set the high address in the program segment, or to the base of the transient portion of COMMAND.COM, whichever is the lower. The contents of the word at Base Page offset 6 are decremented by 1ØØhex to allow for a stack of that size.

A word of zeros is placed at the top of the stack.

The Base Page

The Base Page is structured as follows - with offsets in Hex

Offset	Contents
0000	INT 20hex. Word.
0002	Total Memory size in paragraph form (i.e. 2000hex is equivalent to 256k bytes). Word.
0005	Far CALL to MS-DOS function dispatcher. 5 bytes.
000A	Program Terminate address as IP and CS. 2 words.
000E	Control Break address as CS and IP. 2 words.
005C	File Control Block #1, formatted as normal unopened FCB. 8 words.
006C	File Control Block #2, formatted as normal unopened FCB. 8 words.
0080	Count of characters on command line; followed by command line entered. This region may be used as disk transfer address.

Normal File Control Block

The normal file control block is structured as follows - with offsets in decimal:

Byte	Contents
Ø	The drive number. The drives are numbered as follows: Before opening file: Ø=default drive 1=drive A 2=drive B 3=drive C, etc After opening file: 1=drive A 2=drive B, etc MS-DOS replaces the default drive prefix of Ø with the correct drive number after the open is processed.
1-8	Filename, left justified with trailing ASCII space(s). If a device name is placed in this region, the trailing colon should be omitted.
9-11	Extent, left justified with trailing ASCII space(s).
12-13	Current block number relative to the beginning of the file, starting with zero (automatically set to zero by the open function request). A block consists of 128 records, each record being of the size specified in the logical record size field. The current block number is used with the current record field for sequential reads/writes.
14-15	Logical record size in bytes. Set to 8Øhex by the open function request.
16-19	File size in bytes. The first word represents the low-order part of the file size.

20-21 Date the file was created or last updated. The date is set by the open function request. The date is formatted as follows:

```
<                  21          > <          20              >
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
y  y  y  y  y  y   y  m  m  m  m  d  d  d  d  d
```

where	m	month	1thru12
	d	day	1thru31
	y	year	0thru119 (1980thru2099)

22-23 Time the file was created or last updated. The time is set by the open function request. The time is formatted as follows:

```
<                  23          > <          22              >
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
h  h  h  h  h  m   m  m  m  m  m  s  s  s  s  s
```

where	h	hours	0thru23
	m	minutes	0thru59
	s	seconds*2	0thru59

24-31 Reserved for system use.

32 Current relative record number (0-127) within the current block. This must be set before doing seqeuntial read/write operations on the file. The open function request does not set this field.

33-36 Relative record number, relative to the origin of the file, starting at zero. This field must be set prior to doing random read/write operations on the file. The open function request does not set this field.

If the record size is less than 64 bytes, both words are used. If the record size is greater than 64 bytes, then only the first three bytes are used.

Notes: The File Control Block at 5Chex in the Base Page overlaps both the File Control Block at 6Chex and the first byte of the command line area/disk transfer area at 80hex.

Bytes 0thru15 and 32thru36 must be set by the user program. Bytes 16thru31 are set by MS-DOS and may only be changed at the programmers own risk.

In the 8086/8088 all word fields are stored least significant byte first - this is true in setting the record length, etc.

Extended File Control Block

The extended FCB is used to create or search for files having special attributes. The extended FCB adds an additional 7 bytes preceeding the normal FCB. The extended FCB is structured as follows:

Byte	**Contents**
FCB-7	Set to FFhex indicates that an extended FCB is being used.

FCB-6 to FCB-2 are reserved.

FCB-1	Attribute byte to include hidden files (Ø2hex) or system files (Ø4hex) in directory searches.
FCB-Ø	Origin of normal FCB (drive byte).

APPENDIX I

I.1 Interrupt Driven Serial Input/Output

This appendix is designed to show the methodology involved in driving the Victor 9ØØØ in interrupt mode when communicating via the serial port(s). Some pitfalls are described, and tested sample routines are included. There are, currently, no system level facilities that enable this task to be accomplished easily, and some chips, namely the PIC 8259, PIT 8253, SIO 72Ø1 and the VIA 6522 will require re-programming. It is up to the programmer to reset the machine to the original state prior to exiting the interrupt driven application.

A typical interrupt driven application will normally follow the steps outlined below:

1) Save the original vector, set the new vector.
2) Set the direction bits.
3) Enable clocks (internal or external).
4) Reset SIO 72Ø1 device, define your communication characteristics.
5) Set the baud rate.
6) Set the PIC 8259 to enable SIO interrupts.

These steps will be discussed in more detail throughout the text.

I.2 Interrupt Vectors

There are 256 software interrupts available to the Victor 9000. Most are reserved for system functions, and diagnostics. A block of vectors from 80Hex thru BFHex are set aside for applications.

I.2.1 Vectors available on the Victor 9000

00 - 1FHex	Intel reserved.
20 - 3FHex	Microsoft reserved.
40 - 7FHex	Victor reserved.
80 - BFHex	Applications reserved.
C0 - FFHex	Victor reserved.

Vectors 40Hex thru 47Hex are those belonging to devices controlled by the Programmable Interrupt Controller (PIC).

40Hex	Sync IRQ
41Hex	SIO 7201
42Hex	Timer 8253
43Hex	General Interrupt Handler (all 6522 IRQ's)
44Hex	IRQ4
45Hex	IRQ5
46Hex	Keyboard - keystroke
47Hex	8087 math processor

I.2.2 Location of Vectors

Vectors consist of a long pointer (double word) to an interrupt service routine. This pointer is a 4 byte entry consisting of the Segment and Offset of the Interrupt Service Routine. The vectors are stored in a table that has its origin at 0000:0000. The first entry in this table is, therefore, Interrupt 0; the vector for Interrupt 1 is the second, with its vector having an origin of 0000:0004. The interrupt vector for Interrupt 41Hex (the SIO 7201) will be found at location 0000:0104 (4*41Hex).

To set a vector into this table, the MS-DOS function 25Hex can be used, but since it is desirable to restore the old vector prior to the application program exiting, it is less cumbersome to simply set the new vector "by hand", and restore the old vector when the application terminates.

I.2.3 Set Vector - Assembler Example

```
;store old vector, and set new vector for SIO

     cli                                    ;clear interrupts
     xor  ax,ax                             ; AX = 0000
     mov  ES,ax                             ;access table via ES
     mov  ax,word ptr ES:[104h]             ;get old offset
     mov  word ptr old_offset,ax            ;save old offset in DS
     mov  ax,word ptr ES:[106h]             ;old segment
     mov  word ptr old_segment,ax           ;save old segment
     mov  ax,my_sio_isr                     ;get offset to my code
     mov  word ptr ES:[104h],ax             ;set vector offset
     mov  word ptr ES:[106h],CS             ; and the new segment
     sti                                    ;enable interrupts
     ret                                    ;all done, exit

;to replace the old vector prior to exit

     cli                                    ;clear interrupts
     xor  ax,ax                             ; AX = 0000
     mov  ES,ax                             ;access table via ES
     mov  ax,word ptr old_offset            ;get old offset
     mov  word ptr ES:[104h],ax             ;restore old offset
     mov  ax,word ptr old_segment           ;get old segment
     mov  word ptr ES:[106h],ax             ;restore old segment
     sti                                    ;enable interrupts
     ret                                    ;all done, exit
```

I.3 Enabling Internal and External Clocks

In an asynchronous environment the transmit clock is generated internally, as opposed to a synchronous environment where the transmit clock is typically provided by an external source.

Internal clocking is selected by masking off the appropriate bit in register 1 of the keyboard Versatile Interface Adaptor (VIA).

The keyboard VIA, resgister 1, is located at E8Ø4:ØØØ1.
The appropriate bits are:

Bit Ø (PAØ) for port A
Bit 1 (PA1) for port B

Thus, by setting PAØ to zero, the internal clock is enabled for port A; setting PA1 to zero will enable the internal clock for port B. Setting PAØ or PA1 to one will enable the external clock, disabling the internal clock. CAUTION: Care must be taken to leave the other bits in the pre-selected state.

To enable internal clocks for ports A and B mask off the two least significant bits in register 1:

```
mov  ax,Øe8Ø4h                 ;keyboard VIA segment
mov  ES,ax                     ;select the segment register
and  byte ptr ES:[ØØØ1],Øfch   ;A & B internal clocks done
```

To enable external clocks on either channel then set the relevent bit by OR'ing the bit in. The following sample sets the external clocks for both ports A and B:

```
mov  ax,Øe8Ø4h                 ;keyboard VIA segment
mov  ES,ax                     ;select the segment register
or   byte ptr ES:[ØØØ1],Ø3h    ;A & B external clocks done
```

I.3.1 Providing Clocks

In a synchronous environment it sometimes becomes necessary to provide transmit and receive clocks from the Victor 9ØØØ. This requires that the cable used has pins 15, 17 and 24 jumpered at the Victor 9ØØØ end. The Victor 9ØØØ always has a clock on pin 24, this being provided by the internal baud rate generator; thus by jumpering pin 24 to both pins 15 and 17, this clock becomes available for both the transmitter and the receiver, at both ends of the cable.

When providing clocks from the Victor 9ØØØ, the external clock must be set as well as a baud rate selected. In synchronous mode, the "divide by rate" of the PIT 8253 is 1, therefore the values used to set the required baud rate is 1/16 the values used in an asynchronous environment. (See section 3.8.2 for values).

I.4 Initializing the SIO

There is little magic used in this step, but it is recommended that the programmer read the entire Intel/NEC 72Ø1 chip data sheet. The SIO segment is found in segment location EØØ4Hex. The offsets for the data ports A and B and control ports A and B are at Ø, 1, 2, 3 respectively.

The following example of initializing the SIO 72Ø1 is for Port A:

```
     cli                                ;disable interrupts
     mov  ax,Øe0Ø4h                     ;the SIO segment
     mov  ES,ax                         ;  using ES
     mov  byte ptr ES:[ØØØ2h],18h       ;channel reset

; now delay at least 4 system clock cycles

     nop
     nop                                ;delay for 72Ø1

     mov  byte ptr ES:[ØØØ2h],12h       ;reset external/status
                                        ;  interrupts

;and select register 2

     mov  byte ptr ES:[ØØØ2h],14h       ;non-vectored
     mov  byte ptr ES:[ØØØ3h],Ø2h       ;select CR2 B
     mov  byte ptr ES:[ØØØ3h],ØØh       ;set vector to Ø

; set for clock rate of 16*; 1 stop bit; parity disabled

     mov  byte ptr ES:[ØØØ2h],Ø4h       ;select CR4 A
     mov  byte ptr ES:[ØØØ2h],44h       ;

; this register defines the operation of the receiver:
; 7 data bits; auto enable and receive enable

     mov  byte ptr ES:[ØØØ2h],Ø3h       ;select CR3 A
     mov  byte ptr ES:[ØØØ2h],61h       ;

; CR5 controls the operation of the transmitter
; 7 data bits, dtr; assumes half-duplex

     mov  byte ptr ES:[ØØØ2h],Ø5h       ;select CR5 A
     mov  byte ptr ES:[ØØØ2h],ØaØh      ;

; set status:  affects the vector, interrupt on every character,
; enable transmitter interrupt

     mov  byte ptr ES:[ØØØ2h],Ø1h       ;select CR1 A
     mov  byte ptr ES:[ØØØ2h],17h       ;

     sti                                ; enable interrupts
```

I.4.1 Baud Rate for SIO

At this point, baud rate must be selected. In an asynchronous environment the PIT 8253 divides the supplied baud rate by 16; but in a synchronous environment the baud rate is divided by 1. Thus, to set the baud rate in an asynchronous environment, the value written to the PIT 8253 is 16 times the desired baud rate value. The common baud rate values, and the method of establishing the baud rates, are shown in Section 3.8.2 of this manual.

I.4.2 Set the PIC to Enable SIO Interrupts

In the Victor 9ØØØ the PIC is normally initialized to operate the SIO in a polled environment. The following lines of code sets the PIC to operate the SIO in an interrupt environment:

The PIC resides at segment EØØØHex and the register required here is at offset ØØØ1:

```
cli                          ;disable interrupts
mov  ax,ØeØØØh               ;get the PIC segment
mov  ES,ax                   ;
and  byte ptr ES:[ØØØ1h],(not Ø2h) ;mask off bit 1
.
.
sti                          ;allow interrupts
```

Prior to exiting the interrupt drievn application, the PIC should be returned to operating the SIO in polled mode. This is done by setting bit 1:

```
cli                          ;disable interrupts
mov  ax,ØeØØØh               ;get the PIC segment
mov  ES,ax                   ;
or   byte ptr ES:[ØØØ1h],Ø2h ;set polled
sti                          ;allow interrupts
```

I.5 Interrupt Service Routine - ISR

When an interrupt occurs in non-vectored mode, SIO register CR2 B contains the vector number of the interrupting device. Assuming the SIO was initialized as earlier described in this appendix, CR2 B contains a value in the range Ø-7, which serves as the index to the following interrupt vector table:

I.5.1 Sample Interrupt Service Routine

```
data segment   public    'data'
int_vectors    dw   tx_int_b            ;tx int for port B
               dw   ext_status_b        ;external status changed
               dw   recv_int_b          ;recv int port B
               dw   recv_err_b          ;recv error port B
               dw   tx_int_a            ;tx in for port A
               dw   ext_status_a        ;external status changed
               dw   recv_int_a          ;recv in port A
               dw   recv_err_a          ;recv error port A
data ends

code segment   public    'code'
     assume    CS:cgroup, DS:dgroup

sio_isr:
     mov  word ptr CS:current_ss,SS     ;save stack seg
     mov  word ptr CS:current_sp,SP     ; and stack pointer
     mov  SS,word ptr CS:ss_origin      ;internal stack
     mov  SP,offset dgroup:stack_top    ; defined in DS (dgroup)

     push ax                            ;save environment
     push bx
     push cx
     push dx
     push bp
     push DS
     push ES

     mov  DS,dgroup                     ;set to internal data
     mov  ax,0e004h                     ;set SIO segment
     mov  ES,ax                         ;
     mov  byte ptr ES:[0003h],02h       ;select CR2 B
     mov  al,ES:[0003h]                 ;read int device
     add  al,al                         ;word align
     mov  ah,0                          ; hi = 0
     mov  bx,offset int_vectors         ;get vector table
     add  bx,ax                         ;point to entry
     call [bx]                          ;service routine
     cli                                ;keep disabled
```

--See next page for continuation--

I.5.1 continued

```
; now an "end of interrupt" (EOI) must be issued to the
; SIO (port A) and to the PIC.

        mov  ax,0e000h                          ;PIC segment
        mov  DS,ax                              ;
        mov  byte ptr [0042h],38h               ;EOI to ctrl A of SIO
        mov  byte ptr [0000h],61h               ;EOI to PIC ctrl port A

        pop  ES                                 ;restore environment
        pop  DS
        pop  bp
        pop  dx
        pop  cx
        pop  bx
        pop  ax

        mov  SS,word ptr CS:current_ss          ;get SS
        mov  SP,word ptr CS:current_sp          ;get SP
        iret                                    ;interupt return

; the SS origin is stored here during initialization
ss_origin       dw   0                          ;stack segment origin
current_sp      dw   0                          ;SP on ISR entry
current_ss      dw   0                          ;SS on ISR entry
```

NOTE: Some variables are stored within the code segment, as the CS register is the only register containing a known value at the time of interrupt.

I.6 Setting Direction Bits

This function need only be performed once, and is performed by the operating system BIOS following a hardware reset. This step need not be implemented, therefore, if a standard Victor or Sirius operating system is used. If a standard operating system is not used, then this step needs to be performed immediately prior to the enable clock code.

```
;The offset to the data direction register is ØØØ3Hex.

     cli                                  ;disable interrupts
     mov  ax,Øe8Ø4h                       ;kbd VIA segment
     mov  ES,ax                           ;
     mov  al,byte ptr ES:[ØØØ3h]          ;get the old value
     or   al,Ø3h                          ;set for output
;
; set the PA2-5 to zero, to enable DSR and RI input
;
     and  al,Øc3h                         ;mask in
     mov  byte ptr ES:[ØØØ3h],al          ;rewrite new value
     .
     .
     sti                                  ;enable interrupts
```

APPENDIX J

J.1 Character Set Header

All files with the extension .CHR are Character Set table files. These files contain data corresponding to the actual dot matrix displayed for each character on the console. These files also contain information regarding the character set name, version number, origin, date of creation, and display class. The Character Set table file header is a 128 byte field, structured as follows:

Byte No. Hex	Dec	**Function**
ØØ	ØØ	Character Set type, ASCII 'C' = character
Ø1	Ø1	Character Set Version Number (ASCII Ø thru 9)
Ø2-ØD	Ø2-13	Display Class
ØE-15	14-21	Character Set Name
16	22	Filler (ASCII Space)
17-19	23-25	Banner Class
1A	26	Filler (ASCII Space)
1B-3D	27-61	Comment
3E-4D	62-77	Originator
4E-55	78-85	Creation Date - arranged as YY/MM/DD
56-59	86-89	Number of records in the file in ASCII. A character set file of 128 characters has 32 records; a character set file of 256 characters has 64 records. The record count for a 32 record file is stored as 3Ø 3Ø 33 32 (ØØ32).
5A-5B	9Ø-91	Reserved.

Over...

Byte No. Hex	Dec	Function
5C	92	This byte is used to house three variables. Bit 7 is used to show the Horizontal/Vertical alignment of the character set - bit 7 ON infers a Vertical character set. Bits 6 thru 4 of the high nibble is used to store the binary Super/Subscript value (which may be 1 thru 7) offset from 1 - thus a Super/Subsript value of two would be stored as binary 2. The low nibble is used to store the binary Character Height offset from Ø - thus a Character Height value of 16 would be stored as binary F. The Character height is a function of the number of vertical pixels the character will occupy in the 16x1Ø pixel matrix available for each character on the screen. If the Horizontal/Vertical bit, the Super/Subscript value and the Character Height value was as stated above, then this byte would read AF. The byte appears:

```
    Bit          [7]          [ 6 5 4 ]        [ 3 2 1 Ø ]
    Function     Horiz/Vert   Super/Sub        Character Height
```

Byte No. Hex	Dec	Function
5D	93	This byte contains two values; the User/System character set toggle, bit Ø stores this value; and the Stock/Special character set toggle, bit 1 stores this value. Bit Ø ON infers that the character set is a system character set. Bit 1 ON infers that the character set is a special character set.
5E	94	This byte contains information on the character set width. If the high nibble is Ø, then the low nibble contains the binary information, offset from Ø, of all the characters in the character set - thus a character set width value of 16 would be stored as F. If the high nibble is F, then the character set is a proportional one - the proportional character set has a trailing record containing information on the width of each individual character in the character set. A proportional character set is designed to be used in high-resolution mode as it requires a 16x16 screen cell.
5F-7F	95-127	Reserved.
8Ø-	128-	The character set font information.

Sample Character Set Table File Header

Following is an actual header taken from the Character Set Table file for the character set PROP.CHR. PROP contains 128 characters, and is a proportional character set:

```
 Hex Offset                     Value in Hex

     Ø:       43 3Ø 49 6E 74 27 6C 2Ø   2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 5Ø 52
    1Ø:       4F 5Ø 2Ø 2Ø 2Ø 2Ø 2Ø 43   48 52 2Ø 54 68 69 6E 2Ø
    2Ø:       7Ø 72 6F 7Ø 6F 72 74 69   6F 6E 61 6C 2Ø 63 68 61
    3Ø:       72 61 63 74 65 72 2Ø 73   65 74 2Ø 2Ø 2Ø 2Ø 53 69
    4Ø:       72 69 75 73 2Ø 53 79 73   74 65 6D 73 2Ø 2Ø 38 32
    5Ø:       2F 3Ø 37 2F 31 36 3Ø 3Ø   33 3Ø ØØ ØØ 7F ØØ FF ØØ
    6Ø:       ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
    7Ø:       ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
```

J.2 Proportional Character Set Trailer Information

In the case of a proportional character set, the trailing 128 bytes of the character set file contains information on the proportional width of each of the characters in the file. A proportional character set may not, therefore, contain more than 256 characters.
The following is a sample taken from the character set PROP.CHR; the hex figures represent the width for each proportional character starting with the space character. Note that each width value is offset from Ø, with a value range of 1 thru 16 decimal. Each byte is stored, and represented below, in low/high order; the two nibbles would be exchanged to give the value to the character(s) in high/low order. Each character is mapped from the proportional width as follows:

```
29 95 98 49 77 88 84 93
```

The above figures are for the first 16 display characters including the space character - they correspond as follows:

space	= 1Ø	(corresponding to 9)
!	= 3	(corresponding to 2)
"	= 6	(corresponding to 5)
#	= 1Ø	(corresponding to 9)
$	= 9	(corresponding to 8)
%	= 1Ø	(corresponding to 9)
&	= 1Ø	(corresponding to 9)
'	= 5	(corresponding to 4)
(	= 8	(corresponding to 7)
)	= 8	(corresponding to 7)
*	= 9	(corresponding to 8)
etc		

J.3 **Keyboard Table Header**

All files with the extension .KB are Keyboard Table file. These files contain information regarding keyboard code generated when a key on the keyboard is pressed. These files also contain information regarding the Keyboard Table name, version number, origin, date of creation, and display class. The Keyboard Table table file header is a 128 byte field, structured as follows:

Byte No.		**Function**
Hex	**Dec**	
ØØ	ØØ	Keyboard table type, ASCII 'K' = character
Ø1	Ø1	Keyboard table Version Number (ASCII Ø-9)
Ø2-ØD	Ø2-13	Display Class
ØE-15	14-21	Keyboard Table Name
16	22	Filler (ASCII Space)
17-19	23-25	Banner Class
1A	26	Filler (ASCII Space)
1B-3D	27-61	Comment
3E-4D	62-77	Originator
4E-55	78-85	Creation Date - arranged as YY/MM/DD
56-59	86-89	Number of records in the file in ASCII. A character set file of 128 characters has 32 records; a character set file of 256 characters has 64 records. The record count for a 32 record file is stored as 3Ø 3Ø 33 32 (ØØ32).
5A-7F	9Ø-127	Reserved.
8Ø-	128-	Keyboard table information.

J.4 Banner Skeleton Files

Files with the extension .BAN are banner skeleton files. The banner is information printed on the screen during system boot. The banner also prints the Logo (if selected) along with other information regarding configuration. The banner is a set of ASCII strings containing the escape sequences and characters necessary to print the logo and configuration information on the console.

The first 128 bytes of the Banner Skeleton has the following format. The first byte is zero followed by ØDh, ØAh. This is followed by the length of the file in ASCII decimal with a leading and trailing space, and followed by ØDh, ØAh.

The location of the keyboard name and character set name follow in the same format as the file name length. If the file length is 639 characters, the keyboard name is at byte 5Ø2, and the character set name is at 541, then the first 24 bytes of the banner file would be as follows:

```
30 0D 0A 20 36 33 39 20 0D 0A 20 35 30 32 20 0D 0A 20
35 34 31 20 0D 0A
```

APPENDIX K

K.1 Victor 9000 Disk Structure

K.1.1 Victor 9000 Floppy Disk Structure

The Victor 9000 disk system requires that each track has a variable number of sectors, with each sector containing 512 bytes, with 4 sectors per Allocation Unit (AU), the track structure is as follows:

Track Format

	Track Numbers			
Zone Number	Lower Head (*)	Upper Head	Sectors Per Track	Rotational Period (MS)
0	0-3	(unused)	19	237.9
1	4-15	0-7	18	224.5
2	16-26	8-18	17	212.2
3	27-37	19-29	16	199.9
4	38-48	30-40	15	187.6
5	49-59	41-51	14	175.3
6	60-70	52-62	13	163.0
7	71-79	63-74	12	149.6
8	(unused)	75-79	11	144.0

Notes:
(*) The upper head is not present on the single-sided floppy machine; only the double-sided floppy machine has the upper and lower heads as specified in the table.

MS-DOS allocates space on a Single Sided diskette (SS) and a Double Sided (DS) diskette as follows:

Track 0 Sector 0	Disk Label
Track 0 Sectors 1-2	Two copies of the File Allocation Table (FAT), one FAT in each sector. (SS).
Track 0 Sectors 1-4	Two copies of the FAT, two sectors per FAT. (DS).
Track 0 Sectors 3-10	Directory (SS)
Track 0 Sectors 5-12	Directory (DS)
Track 0 Sectors 11-	Data Region (SS)
Track 0 Sectors 13-	Data Region (DS)

Files, under MS-DOS, are not necessarily written sequentially on the diskette. Diskette space for a file in the data region is allocated on a sector by sector basis, skipping any currently allocated sectors. The first unused sector found in the data region will be the next sector used, regardless of where it appears on the diskette, This method allows for the most efficient use of the disk space available, as sectors made available once a file has been erased can be re-allocated to new files.

K.1.2 Victor 9ØØØ Hard Disk Structure

The hard disk system, in the Victor 9ØØØ, is split into virtual volumes - thus what is in reality one physical disk may be broken into several virtual disks. This means that one large disk system is broken up into several smaller, and therefore, more managable smaller 'disks'.

The virtual volumes are described by a volume list placed in the drive label by the hard-disk configuration utility. This list could be of any length, but in practice will contain only a few entries. Partitioned into smaller 'disks', where each hard disk partition will appear as contiguous storage to the user; this is achieved by dividing the physical address space into Regions and translating logical addresses into these areas. Regions typically represent usable areas between unusable spots in the media. The initial Region list is created after the unit is formatted and configured, and it is ordered by physical address. If areas of the disk should become 'bad' during use, the list can be re-ordered to effectively replace the bad track with a spare track located elsewhere on the disk.

K.1.2.1 Victor 9000 Hard-Disk Label Format

The hard-disk has a label that is used both at boot and run time, this label informs the system of the size and structure of the hard-disk media. Located in sector 0, the label is as follows:

Field Name	Data Type	Contents	BOOT	BIOS	HDSETUP	TEST
Label_Type	WORD	0000 = unqualified 0001 = Current Rev.	R	R	R/W	O
Device_ID	WORD	0001 = Current Rev.	R	R	R	W
Serial_Number	BYTE(16)	ASCII	-	-	R	W
Sector_Size	WORD	512	R	R	R	W
IPL_Vector						
Disk_Address	DWORD	Logical Address	R	-	W	O
Load_Address	WORD	Paragraph Number				
Load_Length	WORD	Paragraph Count				
Cod_Entry	PTR	Memory Address				
Primary_Boot_Volume	WORD	Virtual Volume #	-	R	W	O
Control_Parms	BYTE(16)	(for Tandon TM603SE)	R	R	-	W
# Cylinders	BYTE(Hi)	00Hex				
	BYTE(Lo)	E6Hex (=230)				
# Heads	BYTE	06Hex (=6)				
1st reduced-	BYTE(Hi)	00Hex				
current cyl.	BYTE(Lo)	80Hex (=128)				
1st write-	BYTE(Hi)	00Hex				
precomp cyl.	BYTE(Lo)	80Hex (=128)				
ECC data burst	BYTE	0BHex (=11)				
Options	BYTE	02Hex (=2)				
Interleave	BYTE	05Hex (=5, note that 0 means 5)				
Spares	BYTE(6)	00Hex				
Available_Media_List			-	-	R	W
Region_Count	BYTE	Number of Regions				
Region_Descr	(var)	(Variable by Region_Count)				
Region_PA	DWORD	Physical Address				
Region_Size	DWORD	Block Count				
Working_Media_List			R	R	R/W	O
Region_Count	BYTE	Number of Regions				
Region_Descr	(var)	(Variable by Region_Count)				
Region_PA	DWORD	Physical Address				
Region_Size	DWORD	Block Count				
Virtual_Volume_List			-	R	R/W	O
Volume_Count	BYTE	Number of Virtual Vols.				
Volume_Address	DWORD	Virtual Volume label Logical Address				

The above table describes those elements found in the hard-disk label, following is a discussion of the meanings of the entries themselves:

- o Label Type - this defines the state of the drive layout and the revision level of the label.
- o Device ID - Classification identifying the arrangement, for example, the drive manufacturer, controller revision number. This allows for the identification of compatible controllers/drives.
- o Serial Number - the serial number of the unit is stored here.
- o Sector size - the physical atomical unit of storage on the media.
- o Initial Program Load Vector (IPL) - this is a descriptor identifying the boot program and its location on disk. This information is generated from the primary boot volume label via the utility HDSETUP.
    - o Disk Address - the logical disk address of the boot program image.
    - o Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load to the highest RAM location.
    - o Load Length - the length of the boot program in paragraphs.
    - o Code Entry - a long memory address of the starting entry of the boot program. Segment of zero defaults to the segment of the loaded program.
- o Primary Boot Volume - the logical address of the virtual volume label containing the IPL vector and configuration information.
- o Controller Parameters - a list of controller dependent information, for use in device reset and configuration.
- o Available Media List - a list of permanent usable areas of the disk. This is derived from the available media list and from the format funciton of the HDSETUP utility.
    - o Physical Address - disk address of the region.
    - o Region Size - the number of physical blocks in the region.

- Working Media List - a list of the working areas of the disk. This is derived from the Available Media List and from the format function of the HDSETUP utility.
    - Physical Address - disk address of the region.
    - Region Size - the number of physical blocks in the region.
- Virtual Volume List - a list of the logical disk addresses of all virtual volume labels.

K.1.2.2 Victor 9000 Hard-Disk Virtual Volume Label Format

The Virtual Volume Label provides information on the structure of the Virtual Volume. Generally the operating system references this label, while the HDSETUP utility will create and reference it. The Virtual Volume Label appears as follows:

Field Name	Data Type	Contents	BOOT	BIOS	HDSETUP	TEST
Label_Type	WORD	0000 = null	-	R	R/W	-
Volume_Name	BYTE(16)	ASCII	-	-	R/W	-
IPL_Vector			R	-	W	O
Disk_Address	DWORD	Virtual Address				
Load_Address	WORD	Paragraph Number				
Load_Length	WORD	Paragraph Count				
Code_Entry	PTR	Memory Address				
Volume_Capacity	DWORD	# of Physical Blocks	-	R	R/W	-
Data_Start	DWORD	Virtual Address	-	R	R/W	-
Host_Block_Size	WORD	MS-DOS = 512 bytes	-	R	R/W	-
Allocation_Unit	WORD	# of Physical Blocks	-	R	R/W	-
Number_of_Directory_Entries	WORD	Entry Count	-	R	R/W	-
Reserved	BYTE(16)	Future Expansion Set to Nulls	-	-	W	-
Configuration_Information			-	R	R/W	-
Assignment_Count	BYTE	# of assignment mappings				
Assignment_	(var)	(Variable by Assignment_Count)				
Device_Unit	WORD	Physical Unit Number				
Volume_Index	WORD	Index into Virtual Volume List				

The above table describes those elements found in the hard-disk Virtual Volume label, following is a discussion of the meanings of the entries themselves:

- Label Type - this defines the type of operating environment that the virtual volume is configured for. It is used for type checking when assigning volumes to drives.

- Volume Name - the name of the virtual volume as defined by the user. It is used for identifying volumes.

- Initial Program Load Vector (IPL) - this is a descriptor identifying the boot program and its location within the virtual volume. This field is used to generate the IPL vector on the drive label when configuring the primary boot volume.

  - Disk Address - the virtual disk address of the boot program image.
  - Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load to the highest RAM location.
  - Load Length - the length of the boot program in paragraphs.
  - Code Entry - a long memory address of the starting entry of the boot program. Segment of zero defaults to the segment of the loaded program.

- Volume Capacity - the number of actual blocks that comprise the virtual volume.

- Data Start - the offset (in blocks) into the virtual volume for the start of data space.

- Host Block Size - the atomical unit used by the host in data trasnsfer operations.

- Allocation Unit (AU) - this operating system dependent field means the storage allocation size used by the host in the virtual volume. It is used in determining disk parameter tables and disk definitions.

- Number of Directory Entries - this operating system dependent field means the number of entries in the hosts directory. It is used in determining disk parameters tables and disk definitions.

- Configuration Information - a list of the drive assignments for a system at boot time. It is used to map logical drives to virtual volumes. This field is referenced via the label of the booted drive.

K.2 MS-DOS Disk Directory Structure

The FORMAT/HDSETUP utilities structure the directory for 128 entries on a floppy diskette, and a user defined number on the hard-disk. The directory entries are structured as follows:

Ø-7	Filename (ØE5Hex in byte Ø indicates that this directory entry is unused).
8-1Ø	Filename extension.
11	File attribute. In MS-DOS 1.25, the contents of this byte may be Ø2Hex indicating a hidden file and Ø4Hex indicating a system file. A directory search will not show files with the above attributes, unless the extended FCB is used. Files without attributes will contain ØØHex in this byte. A file may be made hidden/system only when created.
12-23	Reserved.
24-25	Date when file was created or last updated. The mm/dd/yy are mapped as follows:

```
<          25          > <      24      >
15 14 13 12 11 1Ø 9 8 7 6 5 4 3 2 1 Ø
y  y  y  y  y  y  y m m m m d d d d d
```

where:

yy is a value from Ø-199 (198Ø-2Ø99)
mm is a value from 1-12
dd is a value from 1-31

26-27	Starting AU; the relative AU number of the first block in the file. For file allocation purposes only, relative AU's start at ØØØ. Note that relative AU's ØØØ and ØØ1 are the last two AU's of the directory. Therefore the data region starts at relative AU ØØ2. The relative AU number is stored in normal Intel fashion, Least Significant byte first.
28-31	File size in bytes. The first word contains the low-order part of the size. Both words are stored Least Significant byte first.

K.3 MS-DOS File Allocation Tables

The file allocation table (FAT) is used by DOS to allocate disk space for a file, one sector at a time. The FAT is composed of a 12 bit entry for each Allocation Unit (AU), starting with Track Ø Sector 11 on a single sided disk; Track Ø Sector 13 on a double sided disk, and going through to Track 79 Sector 12 on a single sided disk; Track 158 Sector 11 on a double sided disk.

The third FAT entry (relative AU ØØ2) begins the mapping of the data region; each entry contains three hex digits:

ØØØ	If the AU is unused, and available.
FFF	The last AU in the file.
nnn	Any other hex digits that are the relative AU number of the NEXT AU in the file. The relative AU number of the first AU in the file is kept in the files directory entry.

A copy of the FAT for the last used disk in each drive is kept in RAM, and is written back to the disk whenever the status of the disk space used changes.

APPENDIX L

L.1 Generation of Frequencies with the CODEC

This appendix covers the use of the CODEC chip within the Victor 9ØØØ to generate sound. It is beyond the scope of this text to cover actual human-voice generation, Victor does provide tools to achieve this, but generating a frequency will be discussed.

The CODEC chip generates sounds by producing a wave form; frequency generation is achieved by causing a sine wave to be produced by the CODEC, then varying the time base of the sine wave to create various frequencies. Two steps are involved with frequency generation; first the initialization step. The intialization of the CODEC is to produce the sine wave with no time base, the is achieved by the following code:

```
codec_seg       equ     Øe8ØØh  ;codec chip segment
codec_tab       dw      Ø5EØØH, ØØD4ØH, ØØF8ØH, ØØØCØH
codec_lngth     equ     4       ;4 words in the codec table
ssda            equ     ØØØ6ØH  ;SSDA chip port offset
clkctr          equ     ØØØ8BH  ;Codec clock port
cdclk           equ     ØØØ84H  ;Codec frequency clock

init_codec:
        push    ES
        mov     bx,codec_seg    ;codec chip segment address
        mov     ES,bx           ;ready the segment origin
        mov     bx,ssda         ;point to the serial chip
        mov     si,offset codec_tab ;get the init code
        mov     cx,codec_lngth  ;get the table length value
        cld
;
load_loop:
        lodsw
        mov     ES:[bx],ax      ;save the table value
        loop    load_loop       ;  and loop til CX = Ø
        mov     bx,clkctr
        mov     ES:byte ptr [bx],ØCØH ;enable the CODEC clock
        pop     ES
        ret
```

Once intialized, the CODEC is ready to respond to frequency generation requests. This is simple to achieve by supplying the following subroutine with the correct parameters as follows:

```
      ;     Routine: PLAY_NOTE
      ;     Function: To play a single voice note via the CODEC
      ;     Entries:  CX = Frequency in Hz to be played
      ;               DX = Duration of note in multiples of 2.5mS
      ;               AL=Ø=play note, AL=FF=halt note
play_note:
        or      cx,cx           ;is freq Ø
        jz      dono_end        ;yes, exit
        or      dx,dx           ;is duration Ø
        jz      dono_end        ;yes, exit
dono1Ø:
        push    si
        push    bx
        push    ES
        mov     bx,codec_seg    ;codec chip segment address
        mov     ES,bx           ;ready the segment origin
        cmp     al,ØffH         ;if AL = FF then stop note
        jne     dono5Ø
        mov     word ptr ES:cdclk,Ø ;stop note
        jmp short dono_ret
dono5Ø:
        push    dx              ;save the duration
; Now the input to the SSDA must be calculated - note that the
; following calculation achieves a fairly linear tone generation
;  - any deviance from linearity should be fairly minor due to
;  lack of precision in the divide. The calculation itself is:
;       N=((5ØØ ØØØ/F)/8)-1 ;where F=desired frequency in Hz
;                           ;N is the value for the CODEC clock
; This equation may be broken down to:
;       N=(625ØØ/F)-1
        mov     ax,625ØØd       ;ready LSW of 625ØØ decimal
        xor     dx,dx           ;make MSW zeroes
        div     cx              ;get CODEC input value
        sub     ax,1            ;normalise to desired value
        pop     dx              ;get back duration of note
        mov     bx,cdclk
        mov     ES:[bx],ax      ;give the frequency to clock
time_loop:
        mov     ax,25d          ;ready 1 2.5 millisecond period
micro_loop:
        mov     cl,78h          ;ready the timing value
        shr     cl,cl
        dec     ax              ;1ØØ microseconds has passed - more?
        jnz     micro_loop      ;no, so loop til done
        dec     dx              ;see if the note is finished with
        jnz     time_loop       ; not finished - round again
time_done:          ; note playing is over - flush speaker
        mov     ES:word ptr [bx],Ø      ;clear the speaker to silence
dono_ret:
        pop     ES
        pop     bx
        pop     si                      ;stack clear
dono_end:
        ret                             ; and exit
```